CPU Cache Optimization: Does It Matter? Should I Worry? Why?
An Exploration of the World of CPU Cache Performance

A White Paper by Rogue Wave Software.
May 2011
CPU Cache Optimization: Does It Matter? Should I Worry? Why?

An Exploration of the World of CPU Cache Optimization

by Rogue Wave Software

© 2011 by Rogue Wave Software. All Rights Reserved

Printed in the United States of America

Publishing History:

May 2011

Trademark Information

The Rogue Wave Software name and logo, SourcePro, Stingray, HostAccess, TotalView, IML and PV-WAVE are registered trademarks of Rogue Wave Software, Inc. or its subsidiaries in the US and other countries. HydraExpress, ThreadSpotter, ReplayEngine, MemoryScape, JMSL, JWAVE, TS-WAVE, PyIMSL and Knowledge in Motion are trademarks of Rogue Wave Software, Inc. or its subsidiaries. All other company, product or brand names are the property of their respective owners.

IMPORTANT NOTICE: The information contained in this document is subject to change without notice. Rogue Wave Software, Inc. makes no warranty of any kind with regards to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Rogue Wave Software, Inc. shall not be liable for errors contained herein or for incidental, consequential, or other indirect damages in connection with the furnishing, performance, or use of this material.
# TABLE OF CONTENTS


CPU Cache Optimization: Do I Really Need to Spend Time on This? ................................................. 4
  CPU Cache: The Need to Know. ........................................................................................................... 4
  Our First Example ............................................................................................................................... 5
  Cache Line and Fetch Utilization ........................................................................................................ 5
  More Examples .................................................................................................................................. 7
  Conclusion .......................................................................................................................................... 9

About the Author ................................................................................................................................. 11

About Rogue Wave Software ............................................................................................................. 12
CPU Cache Optimization: Do I Really Need to Spend Time on This?

As developers, we are quite familiar with the concept of “caching”. Operating systems, databases, file systems, even our own software creations all use some sort of cache. But what about CPU caches in modern processors? Are they any different?

This white paper is a must read if you have not considered how memory caches impact coding decisions. This paper does not go through a lengthy description of modern processors, but rather focuses on cache optimization examples to help define realistic expectations regarding “CPU Cache Optimization”.

Caching is done automatically regardless of how you program. However, the questions to ask yourself are: How much faster could your program be if it were written in such a way that cache access is considered and optimized? How many fewer servers will you need? And, is it really a million dollar IT Budget Question?

CPU Cache: The Need to Know.
Caches are used to keep the most frequently used data readily available and easily accessible, increasing overall application performance. In the world of memory hierarchy, a rule of thumb of relative access cost is summarized in this table:

<table>
<thead>
<tr>
<th>Memory System Level</th>
<th>Relative Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
<td>1x</td>
</tr>
<tr>
<td>Higher Cache Levels</td>
<td>10x</td>
</tr>
<tr>
<td>Main Memory</td>
<td>100x</td>
</tr>
</tbody>
</table>

Source: AMD, Michael Wall

What this means is that using data already in a Level 1 (L1) cache is 100 times faster than fetching the data from the main memory. This does not mean that your application runs 100 times faster after you optimize your code, but is a hint as to where to apply your optimization efforts. But the question remains, how much faster can it really get? It is time to consider an example.
Our First Example

Let’s compare two similar C++ programs, Program A and Program B. The only difference is the definition of the structures, with Program A having unused members (c and d) as shown below in Figure 1.

<table>
<thead>
<tr>
<th>Program A</th>
<th>Program B</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>struct DATA {</code></td>
<td><code>struct DATA {</code></td>
</tr>
<tr>
<td><code>  int a;</code></td>
<td><code>  int a;</code></td>
</tr>
<tr>
<td><code>  int b;</code></td>
<td><code>  int b;</code></td>
</tr>
<tr>
<td><code>  int c;</code></td>
<td><code>};</code></td>
</tr>
<tr>
<td><code>  int d;</code></td>
<td><code>};</code></td>
</tr>
<tr>
<td><code>DATA * pMyData;</code></td>
<td><code>DATA * pMyData;</code></td>
</tr>
<tr>
<td><code>for (long i=0; i&lt;10*1024*1024; i++)</code></td>
<td><code>for (long i=0; i&lt;10*1024*1024; i++)</code></td>
</tr>
<tr>
<td><code>{</code></td>
<td><code>{</code></td>
</tr>
<tr>
<td><code>  pMyData[i].a = pMyData[i].b;</code></td>
<td><code>  pMyData[i].a = pMyData[i].b;</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

Figure 1. Programs A and B

It could be concluded that both programs should run at the same speed: both are doing the same exact processing and there is no obvious reason why a modern compiler would not generate the same binary code for the loop. Running the example on a small PC (a Windows 7 machine with a dual core U9400 @ 1.40GHz), with full compiler optimization turned on, one gets the following results:

- Program A executed in 0.16 s.
- Program B executed in 0.08 s

Program B runs in half the time of program A! Let’s dig into the details to see what is happening behind the scenes.

Cache Line and Fetch Utilization

Developers commonly make a key assumption: when loading an “integer” in a cache, we assume that 32 bits get copied from the memory to the CPU’s memory cache. However, these CPU caches do not process data with such a fine level of detail. Instead, they fetch and store memory using fixed size chunks. Regardless if you load a “bit”, an
“integer” or a “long”, a modern CPU such as an Intel Core2 Duo loads not only your data but also the 64 bytes of data around it. This is a “cache line” and a typical cache line is 64 bytes long.

Looking back at Program A in Figure 1, the memory is organized as follows:

![Figure 2. A full 64 byte cache line for Program A. Every block represents a 32 bit integer.](image)

Each fetch from the memory brings one similar cache line to the CPU Cache. However, Program A only uses the ‘a’s and ‘b’s, so only half of the fetched data is actually used. The amount of data brought into the cache line that is actually used is only 50%.

Looking at Program B, the memory is organized as shown in Figure 3:

![Figure 3. A full 64 byte cache line for Program B. Every block represents a 32 bit integer.](image)

Each fetch from the memory brings in only ‘a’s and ‘b’s. Accordingly, the entire fetched data is used. In this case, the fetch utilization is 100%.

Program A brings twice as much data from memory than Program B, but it only uses half of it, even if reading the source code let us believe otherwise. Our benchmark run results confirm the theory with a degradation of performance of about 50%.

This example highlights a crucial point: a program’s efficiency is defined by how well the developer knows memory access paradigms, such as vector, arrays, lists, etc. But in every case, we assume the cost of accessing data is directly proportional to its size. Unfortunately, considering the cache line unit of higher level memory access for the CPU caches adds an extra level of complexity.

Data intensive serial applications must be investigated for CPU cache bottlenecks as soon as the data size exceeds the available cache size. Higher level caches are often shared.
across threads and processes running on the same processor chip, and so the available cache size might be considerably less than the physical cache size. For multithreaded programs on shared memory systems, how the data is moved between caches is often more of an issue than data size.

**More Examples**

The first example demonstrates that cache lines can lead to significant performance degradations if the fetched data is not fully utilized. Unfortunately, visually scanning the code for “unused” data is not always enough to locate such performance degradations.

For example, the following programs in Figure 4 both have the same amount of unused data. However, if we switch just one line of code in the structure declaration, Program D runs 60% faster.

**Program C**
```c
struct DATA {
    char a;
    int b;
    char c;
};
DATA * pMyData;
for (long i=0; i<36*1024*1024; i++)
{
    pMyData[i].a++;
}
```

**Program D**
```c
struct DATA {
    int b;
    char a;
    char c;
};
DATA * pMyData;
for (long i=0; i<36*1024*1024; i++)
{
    pMyData[i].a++;
}
```

*Figure 4. Programs C and D.*

This time, it is caused by the compiler, which places data fields in memory based on their types and sizes. A cache line in Program C looks like the following (white placeholders show memory space wasted due to alignment issues):

*Figure 5. A cache line from Program C.*
In Program D, the memory is organized as follows:

![A cache line from Program D.](image)

Program D is clearly utilizing fetched data significantly better than Program C, matching the gap in performance.

The data structure definition is not always the culprit. For best performance, the key is full utilization of every fetched cache line. Unfortunately, poor access logic can also be a factor and trigger significant performance degradations. Of course, modern processors contain a hardware pre-fetcher, which tries to predict which data is to be used next and bring it into the cache in advance, but they will not cover every scenario.

Here is a good example:

<table>
<thead>
<tr>
<th>Program E</th>
<th>Program F</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char * p;</code></td>
<td><code>char * p;</code></td>
</tr>
<tr>
<td><code>p = new char[SIZE];</code></td>
<td><code>p = new char[SIZE];</code></td>
</tr>
<tr>
<td><code>for (long x=0; x&lt;sRowSize; x++)</code></td>
<td><code>for (long y=0; y&lt;nbRows; y++)</code></td>
</tr>
<tr>
<td><code>{</code></td>
<td><code>{</code></td>
</tr>
<tr>
<td><code>p[x+y*sRowSize]++;</code></td>
<td><code>p[x+y*sRowSize]++;</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

![Figure 7. Programs E and F](image)

Again, these two programs should run at the same speed if we (wrongly) assume that the cost of accessing data is directly proportional to its size. However Program F runs 4 times faster than Program E. The cache line is playing another trick on us. In C++ with row-major memory storage, accessing the data in contiguous rows is four times faster here than accessing it column per column.

The left part of Figure 8 below illustrates what happens when the access is done column by column (Program E). Every time you access a new piece of data, you are loading a new cache
line (the cache line, shown as a blue rectangle, moves down at every memory access).

The alternative (Program F) is pictured on the right in Figure 8. The data is accessed row by row and you only load a new cache line for every four accesses to data.

![Figure 8. Per-column access compared to per-row access.](image)

Ultimately, this access pattern results in Program F running four times faster than Program E due to more efficient usage of the cache.

**Conclusion**

The examples covered only a couple of optimization techniques, focusing on increasing cache line utilization. Ultimately, they show quite substantial gains. There are many other cache related issues not covered, including false sharing, cache thrashing, and non-temporal data issues that can also result in dramatic performance degradation. The examples in this paper simply demonstrate that the cost of accessing data is not directly proportional to its size, and one must recognize the impact memory cache has on data intensive application performance.

These optimization techniques become even more important with the introduction of many-core chips, where more and more competing threads sharing memory access are potentially starved for data. This becomes your application’s next bottleneck.

- Is your code suffering from bad cache utilization?
• Does it need to be optimized?
• Can it be optimized?
• Can your developers make better use of their time?

Rogue Wave Software offers ThreadSpotter, a new tool that is easily integrated in your development workflow to analyze any compiled application for CPU cache concerns.

First, ThreadSpotter provides an overall analysis of your application. This helps you keep track of cache concerns for every build of your product.

![Figure 9. ThreadSpotter provides an overall analysis of any application.](image)

Second, ThreadSpotter also creates a detailed report that shows opportunities for cache optimizations directly in the application source code. Sitting on every developer’s desk, ThreadSpotter validates whether their latest algorithm is “CPU cache efficient”, and explains where improvements can be made.
Finally, ThreadSpotter acts as a mentor and the developers improve their programming skills, writing code that scales better with future many-core hardware. Its comprehensive documentation provides detailed examples and suggestions to work around performance challenges.

About the Author

Stephane Raynaud is a Principal Engineer at Rogue Wave Software with extensive experience in high performance C++ based software architectures. His career at Rogue Wave Software began in 1992 shortly after having emigrated as a computer sciences graduate student to North America. Among Stephane’s many accomplishments are the development of several proof of concept projects related to high performance distributed computing, C++ based SOA, and the delivery of several educational seminars related to coding optimization. His focus on the Financial Services industry has led Stephane to become engaged in a wide range of projects, from quantitative analysis to low-latency systems development, including his most recent work to address advanced CPU Cache optimization for multi-core processors.
About Rogue Wave Software

Rogue Wave Software, Inc. is the largest independent provider of cross-platform software development tools and embedded components for the next generation of HPC applications. Rogue Wave marries High Performance Computing with High Productivity Computing to enable developers to harness the power of parallel applications and multicore computing. Rogue Wave products reduce the complexity of prototyping, developing, debugging, and optimizing multi-processor and data-intensive applications. Rogue Wave customers are industry leaders in the Global 2000, ISVs, OEMs, government laboratories and research institutions that leverage computationally-complex and data-intensive applications to enable innovation and outperform competitors. Rogue Wave is a Battery Ventures portfolio company. For more information, visit http://www.roguewave.com.